Convergence of Rapidly-Exploring Random Trees to Optimal Solutions

Raymond Bjorkman
GRASP Lab
University of Pennsylvania
Philadelphia, PA, USA
raybjork@seas.upenn.edu

***Abstract*—The class of probabilistic planners known as Rapidly-Exploring Random Trees (RRTs) represent a conceptually simple answer to the problem of robot planning. Instead of evaluating the robot's interactions in 3D space we can consider planning in the free configuration space which will be discovered through random sampling. This is an advantage over the classic A* algorithm where this space must be explicitly calculated. For this reason RRT is often simpler to implement and computationally faster than A*. However, RRT also produces trajectories that are usually far from optimal. RRT* is a modification to the base RRT algorithm and is asymptotically optimal but their convergence time can make the algorithm inefficient at finding these solutions. Informed RRT* and RRT*-Smart are two modifications that have been suggested to mitigate this issue. This paper will introduce a new variation to the RRT* algorithm and compare it to existing ones by examining the strengths and limitations of each variation across a variety of test cases.**

I. Introduction

Rapidly-exploring random trees (RRTs) were first introduced to the robotics community in 1998 as a simple solution for robot planning that could be easily extended to take dynamical effects into consideration [1]. Since then, they have been adopted as a solution for a wide variety of problems in robotics. Along with this popularity has come a plethora of innovative variations relying on them as a framework to create more complex algorithms tailored for specific applications.

RRT is a probabilistic planning algorithm, meaning that the solution it finds is one that has been arrived at through random means. A byproduct of this is that it often generates inefficient paths for robots that make them act erratic. There have been a number of proposed methods for mitigating this problem. In 2011 Karaman and Frazolli introduced the algorithm of RRT*. It is able to guarantee asymptotic optimality of the paths that it generates meaning that the solution it returns will converge to the optimal one over time [2].

Unfortunately, the time it takes RRT* to converge to this optimal trajectory can be undesirably long. In the past several years papers introducing two new algorithms, RRT*-Smart and Informed RRT*, were published with the intent to fix this. The approach these methods take broadly a more focused approach by path planning for a specific objective by biasing state space exploration to an area that will improve the path more quickly [3] [4]. This style improves RRT* convergence to the optimal trajectory, but the two algorithms are very different in their implementations of this concept. This paper will first introduce another implementation of this strategy in a new algorithm called Local RRT* which forgoes finding the globally optimal path for finding a locally optimal one very efficiently. This paper aims to compare these three algorithms to each other and the base RRT* algorithm, examining how their performance and computational efficiency change over time. The experimental conditions under which these comparisons will take place will use several different maps specially selected to highlight different strengths and weaknesses of the tested algorithms.

II. Background

All of the algorithms highlighted in this paper will rely on an understanding of the underlying concepts behind the RRT class of planners. At a high level RRTs seek to efficiently explore the free configuration space of a robot without the need to explicitly calculate this region, which is required for other planning algorithms such as A*. Instead, the strategy revolves around randomly sampling state space and checking for local trajectory feasibility between the existing tree and the sampled space.

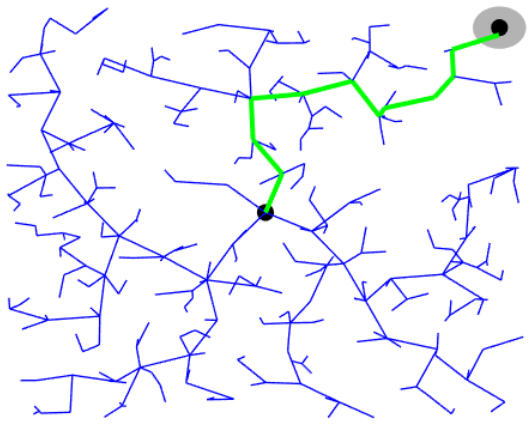


Fig. 1: A simulation of a rapidly-expanding random tree. It grows from the center, sampling state space around it, finds the shaded goal region, and returns a path highlighted in green.

Algorithm 1: RRT(x_{init}, $\mathcal{X}_{goal}$)

$V \leftarrow x_{init}$
$E \leftarrow \varnothing$
for $i = 1, \ldots, n$ **do**
  $x_{rand} \leftarrow$ `Sample()`
  $x_{closest} \leftarrow$ `Nearest_Node`(V, x_{rand})
  $x_{new} \leftarrow$ `Steer(` d_{step}. $x_{closest}$, x_{rand})
  if *Valid_Trajectory*$(x_{new}, x_{closest})$ **then**
    $V \leftarrow V \cup x_{new}$
    $E \leftarrow E \cup (x_{closest}, x_{new})$
    if $x_{new} \subseteq \mathcal{X}_{goal}$ **then**
      return `Shortest_Path`(V, E)
    end
  end
end

Algorithm 1 describes the pseudocode for RRT [**?**]. Given a initial state space and a desired goal region for the system, the RRT algorithm will grow a tree from this initial node. For each random state space sample, it will calculate the closest node that is in the tree. Next it will find a set of inputs that will steer it through state space a distance of `stepsize` in the direction of this random sample. If the trajectory generated to this point is admissible the new node is added to the tree. This process iteratively continues until a state space in the goal region is connected to the tree. The data representation of the tree that gets built here consists of a list of nodes V and an adjacency matrix E.

The algorithm's simplicity has lent itself to modification. The RRT* variations being sampled in this paper will modify the `Sample` function from the default of uniform random sampling to the biasing of certain regions over others. Furthermore, if valid `Nearest_Node`, `Steer` and `Valid_Trajectory` functions can be formulated for the system in question RRT can be adapted for nonholonomic robots. Lavalle's research shows that linear dynamic models can be incorporated into this algorithm by modifying the `Steer` function to integrate the system forward through time a distance d_step for a given input [7]. Other research has looked into alternatives to Euclidean distance as a distance metric for systems where nonlinear dynamics can render it unimportant. Glassman introduced a linear quadratic regulator cost based heuristic method for such systems [5]. This is implemented as the `Nearest_Node` function in a larger RRT used for controlling nonlinear systems such as the cartpole [6]. While this paper will be focused on tests using a holonomic point robot in 2D, it is worth remembering that research into this framework has has exciting and wide reaching implications that extend beyond this simple system.

III. RRT* and Variants

A. RRT*

RRT* modifies the base version of RRT by changing the tree into a weighted graph. The weight of each vertex in the graph can be calculated by a function `Cost`, which for the holonomic point robot in 2D is taken by default to be Euclidean distance along the closest path connecting the vertex to the root of the tree. This cost is used to find the most appropriate node of the tree for the sampled node.

In addition to being the first asymptotically optimal variation on RRT, this algorithm is also probabilistically complete. This means that given an infinite amount of time the expectation value that the map will be fully known is one [1]. Unlike RRT, RRT* is capable of finding paths that are globally optimal even if they are from a different homotopy class than the original solution. A homotopy class is the set of solutions that can be deformed into one another [8]. This concept is important to understand in the context of comparatively evaluating path planning algorithms.

Algorithm 2: RRT*(x_{init}, $\mathcal{X}_{goal}$)

$V \leftarrow x_{init}$
$E \leftarrow \varnothing$
`Cost`$(x_{init}) \leftarrow 0$
for $i = 1, \ldots, n$ **do**
  $x_{rand} \leftarrow$ `Sample()`
  $x_{closest} \leftarrow$ `Nearest_Node`(V, x_{rand})
  $x_{new} \leftarrow$ `Steer(` d_{step}. $x_{closest}$, x_{rand})
  if *Valid_Trajectory*$(x_{new}, x_{closest})$ **then**
    $\mathcal{X}_{neighbors} \leftarrow$ `Near_Valid_Neighbors`$(V,$ $E,$ $x_{new},$ $r)$
    $x_{min} \leftarrow \min x_i \in \mathcal{X}_{neighbors}$ (`Cost`(x_i) + `Cost(Trajectory`$(x_i, x_{new}))$)
    $V \leftarrow V \cup x_{new}$
    `Cost`$(x_{new}) \leftarrow$ `Cost`$(x_{min})+$ `Cost(Trajectory`$(x_{min}, x_{new}))$
    $E \leftarrow E \cup (x_{new}, x_{min})$
    `Rewire`$(V,$ $E,$ $x_{new},$ $\mathcal{X}_{neighbors})$
  end
end
return *Shortest_Path*(V, E)

The pseudocode in Algorithm 2 makes reference to a few other functions that were not present in the base version of RRT. The function `Near_Valid_Neighbors` searches around the queried point for vertices in the graph that lie within in a unit ball with radius r and can also form a valid trajectory to that queried point. The best connection point will be the vertex in the set $\mathcal{X}_{neighbors}$ that has the lowest cost. Later in the algorithm the `Rewire` function checks this set to determine if going through the newly added node is cheaper and updates the cost of the corresponding nodes accordingly [2].

For a planner with uniform sampling these type of changes can take a while to propagate throughout the tree. For example if a shortcut is found the only nodes that will be updated accordingly will be those close to the shortcut. Later in Section IV I will attempt to fix this issue with a sampler optimized for graph information gain.

B. RRT*-Smart

RRT*-Smart attempts to improve convergence of RRT* by effectively jumping unnecessary nodes in the trajectory

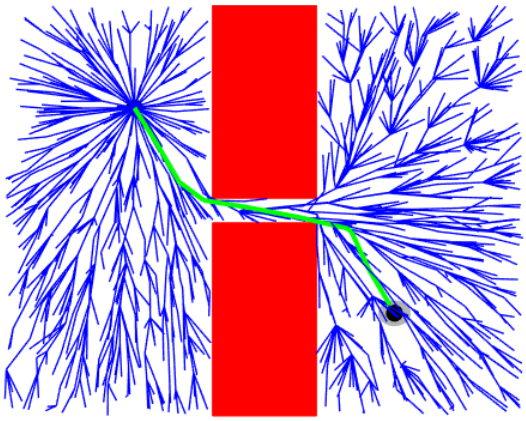


Fig. 2: A simulation of the RRT* algorithm. The fractal patterns it creates are due to it rewiring itself as new nodes are added.

wherever possible. After each new node that is added to the tree an operation, `Smooth_Path` takes place that shortcuts a node along the solution path's parent if that node can connect directly to its grandparent instead. The nodes that are left over after this smoothing takes place are called beacon nodes.

Algorithm 3: RRT*-Smart(x_{init}, $\mathcal{X}_{goal}$)

$V \leftarrow x_{init}$
$E \leftarrow \varnothing$
$c_{min} = \infty$ `Cost`$(x_{init}) \leftarrow 0$
for $i = 1, \ldots, n$ **do**
  $x_{rand} \leftarrow$ `Smart_Sample`$(\mathcal{X}_{beacons}, b(i))$
  $x_{closest} \leftarrow$ `Nearest_Node`(V, x_{rand})
  $x_{new} \leftarrow$ `Steer`$(d_{step}\ x_{closest}, x_{rand})$
  if *Valid_Trajectory*$(x_{new}, x_{closest})$ **then**
    $\mathcal{X}_{neighbors} \leftarrow$ `Near_Valid_Neighbors`(V, E, x_{new}, r)
    $x_{min} \leftarrow \min x_i \in \mathcal{X}_{neighbors}$ (`Cost`(x_i) + `Cost(Trajectory`(x_i, x_{new})))
    $V \leftarrow V \cup x_{new}$
    `Cost`$(x_{new}) \leftarrow$ `Cost`(x_{min})+ `Cost(Trajectory`(x_{min}, x_{new}))
    $E \leftarrow E \cup (x_{new}, x_{min})$
    `Rewire`$(V, E, x_{new}, \mathcal{X}_{neighbors})$
    $c =$ `Smooth_Path`$(V, E, x_{start}, x_{goal})$
    if $c < c_min$ **then**
      $\mathcal{X}_{beacons} =$`Smooth_Path`$(V, E, x_{start}, x_{goal})$
    end
  end
end
return *Shortest_Path*(V, E)

This algorithm then samples based on a bias selected by the designer. When the bias function $b(i)$ is true, instead of sampling uniformly it samples in a unit ball around one of the beacon nodes [3]. This has the effect of shifting the smoothed solution slightly to see if it will improve it.

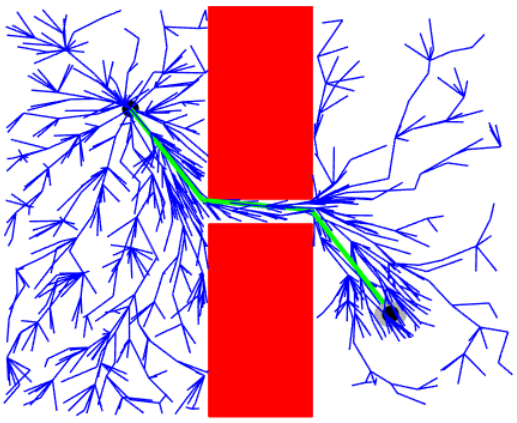


Fig. 3: A simulation of the RRT*-Smart algorithm shows that it samples points along the smoothed path more densely than those on the edges of the map.

One thing to note about RRT*-Smart is that the process of biasing it towards an existing trajectory makes it particularly good at finding locally optimal paths in a single homotopy class. However, this in turn makes it relatively bad at finding a different homotopy class than the original path. It attempts to mitigate this downside by giving the designer control over the bias function, but Gammell notes that this balance violates uniform sampling philosophy of RRT* [4].

Another drawback to RRT*-Smart is the need to run an expensive smoothing subroutine for each newly added node. This adds a constant linear cost to RRT* which is significant when the intent of this algorithm is improved convergence time of RRT*.

C. Informed-RRT*

Informed-RRT* takes a more mathematical approach to biased sampling and reasons that geometrically a solution that will improve the current best path must lie within some ellipse which is defined by the start and end positions and the current best path in the graph between them. By only sampling in this region, Improved RRT* is more likely to find a better solution rather than sampling somewhere that has no bearing on the current objective [4]. This sampling heuristic is referenced as `Sample_Ellipse` in the Informed RRT* pseudocode in Algorithm 4 and also illustrated in Figure 4.

As shown in Algorithm 4 the cost defining the size of the ellipsoidal sampling heuristic only updates when a new node is added within the goal region. This is done to avoid needing to make costly computations to fully calculate the best path through through the graph. As a result, new information can take a long time to be reflected in the heuristic ellipse if a node in the goal region is not sampled for a long time. However, the strength of Informed RRT* is that

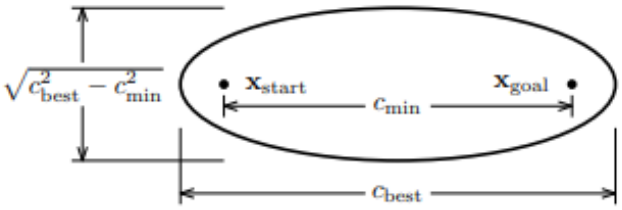


Fig. 4: A diagram illustrating the ellipsoidal sampling heuristic used by Informed RRT*. Image adapted from [4].

even in the worst case it behaves exactly like RRT* [4].

Algorithm 4: Informed RRT*(x_{start}, x_{goal}, $\mathcal{X}_{goal}$)

$V \leftarrow x_{start}$
$E \leftarrow \varnothing$
$\mathcal{X}_{solutions} \leftarrow \varnothing$
$\texttt{Cost}(x_{init}) \leftarrow 0$
for $i = 1, \ldots, n$ **do**
  $c_{best} \leftarrow min x_i \in \mathcal{X}_{solutions}(\texttt{Cost}(x_i))$
  $x_{rand} \leftarrow \texttt{Sample\_Ellipse}(x_{start}, x_{goal}, c_{best})$
  $x_{closest} \leftarrow \texttt{Nearest\_Node}(V, x_{rand})$
  $x_{new} \leftarrow \texttt{Steer}(\ d_{step}.\ x_{closest}, x_{rand})$
  if *Valid_Trajectory*$(x_{new}, x_{closest})$ **then**
    $\mathcal{X}_{neighbors} \leftarrow \texttt{Near\_Valid\_Neighbors}(V,$
      $E, x_{new}, r)$
    $x_{min} \leftarrow \min x_i \in \mathcal{X}_{neighbors}\ (\texttt{Cost}(x_i) +$
      $\texttt{Cost}(\texttt{Trajectory}(x_i, x_{new})))$
    $V \leftarrow V \cup x_{new}$
    $\texttt{Cost}(x_{new}) \leftarrow \texttt{Cost}(x_{min}) +$
      $\texttt{Cost}(\texttt{Trajectory}(x_{min}, x_{new}))$
    $E \leftarrow E \cup (x_{new}, x_{min})$
    $\texttt{Rewire}(V, E, x_{new}, \mathcal{X}_{neighbors})$
    if $x_{new} \in \mathcal{X}_goal$ **then**
      $\mathcal{X}_{solutions} \leftarrow \mathcal{X}_{solutions} \cup x_{new}$
    end
  end
end
return *Shortest_Path*(V, E)

IV. Local RRT*

Learning from the lessons of RRT*-Smart and Informed RRT* I set out to design my own sampling heuristic to use with RRT* with the intent of improving some of the downsides of the algorithms discussed above. One of the main themes of RRT* and Informed RRT* is the unwillingness to propagate information through the tree as costs are updated through rewiring. In the latter case lower latency on this information gain would cause the ellipsoidal sampling heuristic to shrink which would in turn create a ripple effect and accelerate the convergence to the optimal solution. In the case of RRT*-Smart, a costly calculation is repeatedly made in order to not make this sacrifice. I observed that RRT* propagates changes in cost updates through the map coming from the start node and only eventually affecting the outer reaches of the map.

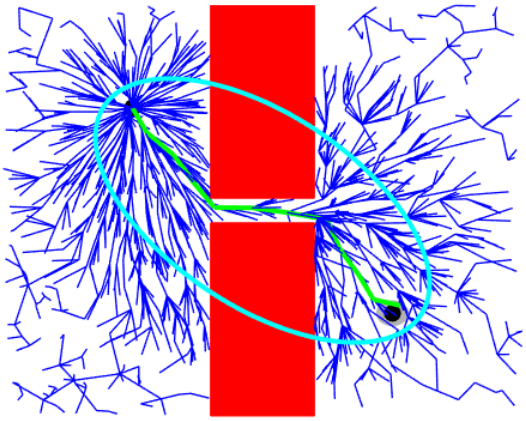


Fig. 5: A simulation of Informed RRT*. As the program continues and the solution improves the cyan ellipse will shrink and the edges of the map will not be sampled as densely as the center.

Algorithm 5: Local RRT*(x_{init}, $\mathcal{X}_{goal}$)

$V \leftarrow x_{init}$
$E \leftarrow \varnothing$
$\texttt{Cost}(x_{init}) \leftarrow 0$
for $i = 1, \ldots, n$ **do**
  $x_{rand} \leftarrow \texttt{Sample\_Propagation}\ (i\%l_{path})$
  if $i + 2\%l_{path} == 0$ **then**
    $l_{path} = \texttt{Length}(\texttt{Shortest\_Path}(V, E))$
  end
  $x_{closest} \leftarrow \texttt{Nearest\_Node}(V, x_{rand})$
  $x_{new} \leftarrow \texttt{Steer}(\ d_{step}.\ x_{closest}, x_{rand})$
  if *Valid_Trajectory*$(x_{new}, x_{closest})$ **then**
    $\mathcal{X}_{neighbors} \leftarrow \texttt{Near\_Valid\_Neighbors}(V,$
      $E, x_{new}, r)$
    $x_{min} \leftarrow \min x_i \in \mathcal{X}_{neighbors}\ (\texttt{Cost}(x_i) +$
      $\texttt{Cost}(\texttt{Trajectory}(x_i, x_{new})))$
    $V \leftarrow V \cup x_{new}$
    $\texttt{Cost}(x_{new}) \leftarrow \texttt{Cost}(x_{min}) +$
      $\texttt{Cost}(\texttt{Trajectory}(x_{min}, x_{new}))$
    $E \leftarrow E \cup (x_{new}, x_{min})$
    $\texttt{Rewire}(V, E, x_{new}, \mathcal{X}_{neighbors})$
  end
end
return *Shortest_Path*(V, E)

With this in mind I propose a new sampling heuristic that samples along the path, on the inside of the triangle created between two lengths of a path. This sampling scheme (`Sample_Propagation` in Algorithm 5) seeks to approximate gradient descent. The logic is that the solution will improve over time if the sample attempts to straighten the line between these paths. Furthermore, this sampling is done sequentially starting from the start node and along the current best path. When it reaches the last node along the

path the algorithm recalculates the best path and starts it over. This seeks to improve information gain of the graph by ”suggesting” that it continue rewiring the cost along the path as new better nodes are found. Furthermore, the expensive recalculation of the best path only happens once in many loops through the code. Figure 6 gives a geometric intuition for how this works in practice.

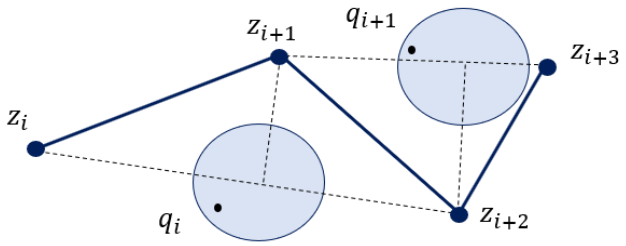


Fig. 6: Sampling scheme for Local RRT*. To keep the cost of the tree globally updated as best as possible unit balls between a node and its grandchild are sampled sequentially. This propagates the rewiring updates through the tree as fast as possible without needing to run computationally expensive routines.

A drawback to this approach is that its narrow sampling domain will make it nearly impossible for it to change homotopy classes to more optimal ones. Rather than try to balance this with global exploration as with RRT*-Smart I propose to only use this algorithm to quickly optimize local trajectories.

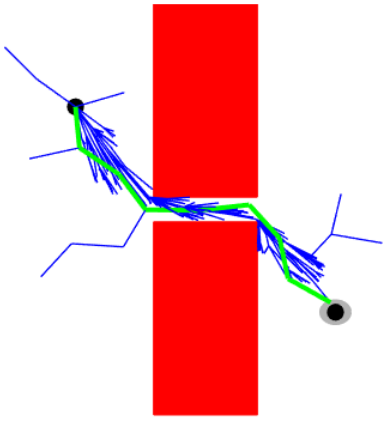


Fig. 7: A simulation of Local RRT*. Rather than sample the entire map, it is only sampling along the best path to try to find the global optimum of the homotopy class of the solution.

V. Experimental Methodology

In order to compare these algorithms to each other I implemented each in MATLAB and simulated their performance in a variety of test conditions.

These test conditions were selected to simulate different types of environments. Map 1 was the empty set, a map devoid of obstacles. Seeing which algorithm performs well on this map will be a good benchmark for robotic systems where few obstacles are expected. Map 2 is the narrow passage map, which has traditionally been a problem for probabilistic planners to navigate. There are globally suboptimal paths that go on either side of the obstacles, but there is also a very low cost shortcut through the middle of the map. This test case will determine how well prepared an optimal planner is to find a difficult solution Map 3 represents a maze that the 2D point robot must navigate. There is only one solution mode to this map and therefore it is a good test of how well each algorithm can optimize a long trajectory. Finally, Map 4 is a randomized forest of 60 obstacles. Unlike the other maps there are many potential solution modes for this map so the planner that performs the best in this map will be the one most well equipped to find the global solution among many homotopy classes.

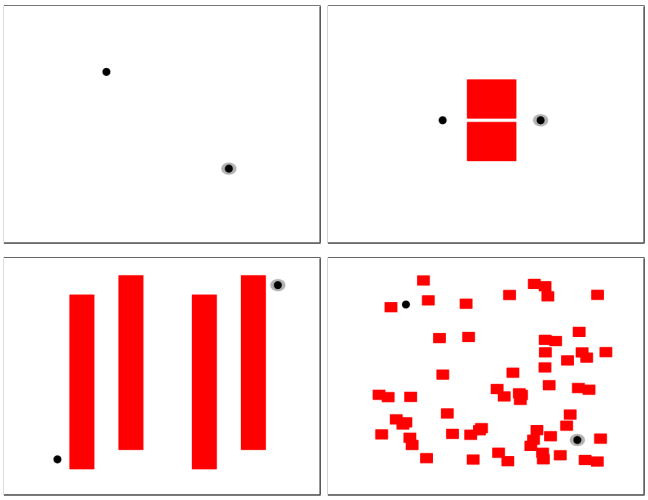


Fig. 8: Maps used to test the variants of RRT*. From left to right, top to bottom: Map 1 - Empty Map, Map 2 - Narrow Passage, Map 3 - Maze, Map 4 - Random Forest.

In order to determine differences in computation time between these algorithms I wrote them so that they shared as much of the same unoptimized code as possible with the only differences coming into play only for the different sampling and smoothing techniques unique to each of them.

Because the algorithms being tested are probabilistic in nature, comparisons between them must be made across a variety of trials. For each map that was tested, I ran 20 trials. For each trial RRT was run to find an initial feasible path from start to end. The graph corresponding to this solution was then passed into each of the four algorithms being tested so that each of the optimizing algorithms was seeded with the same initial path. Each of the four tested algorithms was then allowed to continue on to optimize the path for a specified number of trials. In this way, the effect of randomness on the algorithms was limited to the components I was interested

in testing and was not affected by randomness in initial path generation. Furthermore, design variables inherent to RRT* such as d_{step} were held constant across algorithms

During each simulation the state of the graph was assessed at a frequency of once every fifty sample points. During this assessment the program runtime was not tracked since this interrogation of the system would not be present in a deployment of one of these algorithms. In this way the number of points sampled, the total system runtime in seconds and the cost of the best solution path for the graph (measured in Euclidean distance along the path) were tracked for each simulation.

VI. Results and Discussion

Using the data gathered through the procedure in Section V I tracked both trajectory cost and system runtime as a function of the number of sampled nodes. To get trends for the data, I calculated the mean and one standard deviation of each of the algorithm's performance across the 20 trials for the given map. These metrics are plotted as dotted lines and shaded regions respectively in Figures 9 - 12

The results from the experiments from Map 1 show RRT*-Smart unsurprisingly converging instantly to the optimal trajectory. This is because there are no obstacles in the way so the smoothing algorithm can find the best path immediately. Local RRT* also finds the best path very quickly, with the edges of its standard deviation being well outside that of RRT* and Informed RRT* when the number of nodes is less than a few hundred. There is little difference in the performance of RRT* and Informed RRT*.

Map 2 again shows RRT*-Smart and Local RRT* converging equally quickly to the globally optimal solution of the easy to find homotopy class. At around 500 nodes, Informed RRT* performs best, having found the globally optimal narrow path solution before the others due to its ellipsoidal sampling heuristic. RRT*-Smart and RRT* are then able to find this superior path next at around the same rate, while Local RRT* was unable to ever find it.

In the maze map, Map 3, the situation is similar to the one in Map 1 because again there is only one homotopy class. RRT*-Smart is once again the first to find the optimal path, with Local RRT* coming in a close second. Neither Informed RRT* or RRT* were able to find the optimal solution within the maximum number of samples. The upside to using the ellipsoidal heuristic is diminished in the case of this much larger map.

The first thing to note about the results of the fourth and final map is that the standard deviations for each of the algorithms is very wide. Because this comparative test of probabilistic planners was run on a randomly generated map there is a great deal of variation across these tests. Again, RRT*-Smart and Local RRT* are the quickest to converge to a solution yet, of the two, RRT*-Smart is still able to sample the entire map and therefore is able to navigate the multitude of homotopy classes more easily than Local RRT* which flattens to a locally minimal cost quickly.

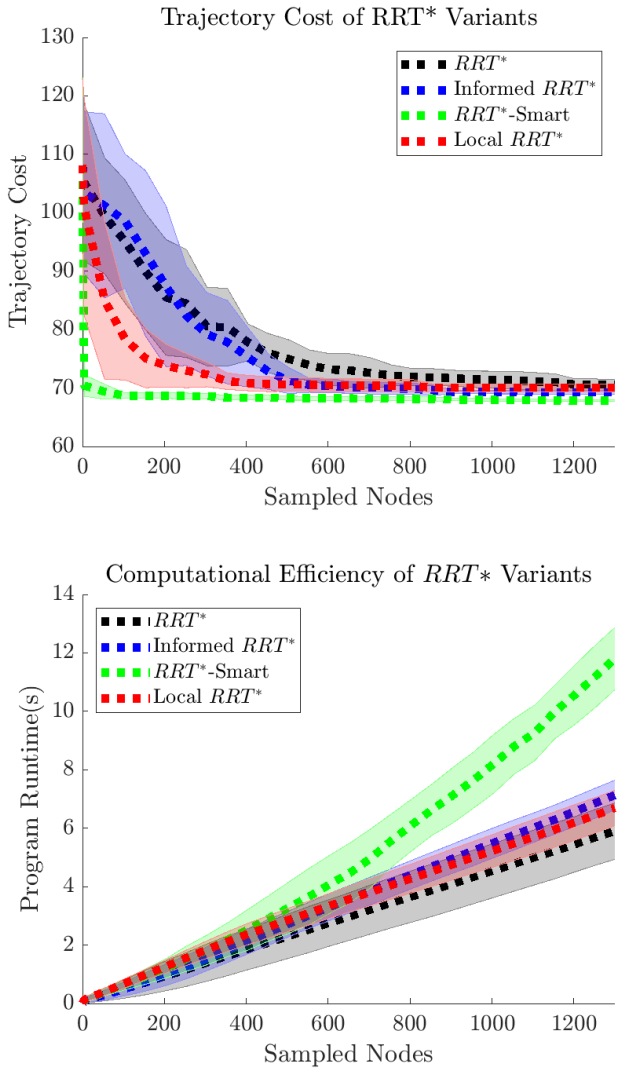
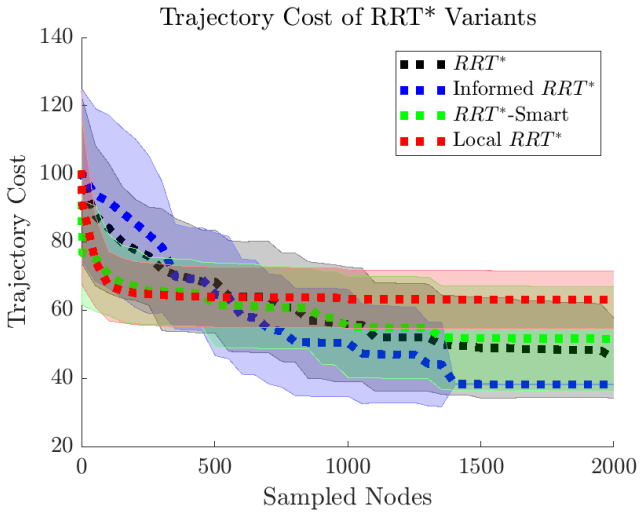


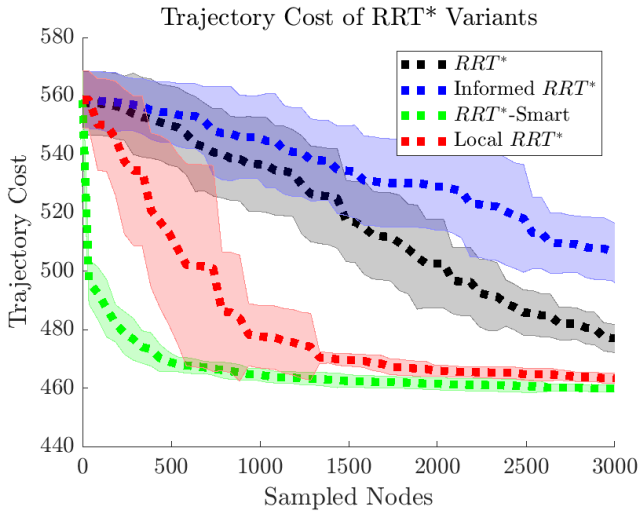
Fig. 9: Map 1 trajectory cost and computational efficiency for RRT* and its variants.

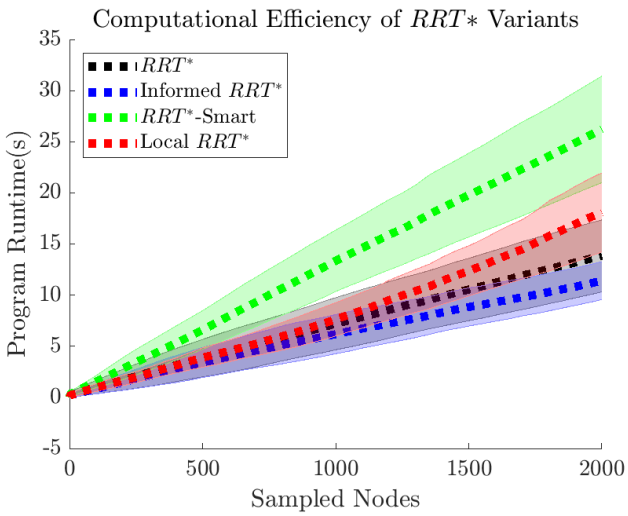
Looking at the computational efficiency of the four test maps it is easy what the disadvantages of RRT*-Smart are. With an increased number of nodes comes an increasingly densely sampled map which requires more rewiring operations alongside the recalculation of the shortest path. Although it does worse than all three other planners in this regard, it in practice still appears to have complexity of $\mathcal{O}(n)$ along with the other algorithms. However, it is possible that the case for using Local RRT* could be made here. On many of the maps Local RRT* performed comparably to RRT*-Smart while using less computing power. For a map with a low number of homotopy classes either option could be a viable choice.

VII. Conclusions

From the simulation of RRT* and three of its variations I have outlined what their strengths and weaknesses are. For







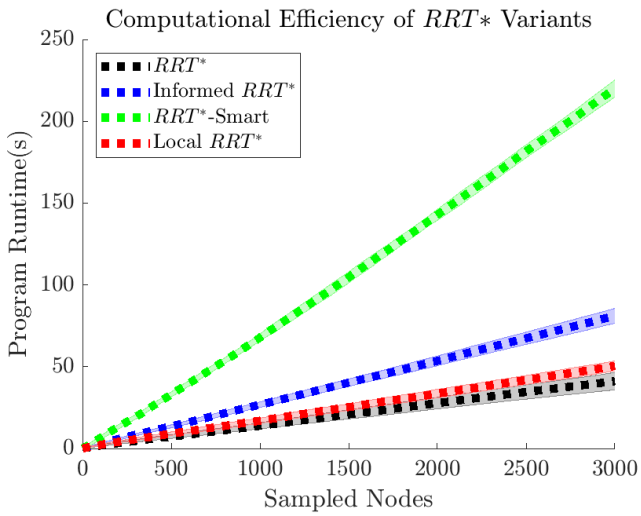


Fig. 10: Map 2 trajectory cost and computational efficiency for RRT* and its variants.

Fig. 11: Map 3 trajectory cost and computational efficiency for RRT* and its variants.

an engineer considering implementing one of the algorithms outlined in this paper, this can be distilled into the takewawys below.

- RRT* is conceptually the simplest to implement and performs well across a variety of maps at the lowest computational cost
- RRT*-Smart is highly efficient at finding optimal trajectories for maps with a low number of homotopy classes and slightly inefficient when there are many. Compared to the other algorithms it has the highest computation cost.
- Informed RRT* is efficient at finding difficult to find homotopy classes and is otherwise comparable to the base RRT* algorithm.
- Local RRT* is eclipsed by RRT*-Smart in nearly every way, except it takes slightly less computing power.

There is still future work to be done in the study of probabilistic planners. I would like to further tune my heuristic sampling approach to see if it can be improved to the point of broader viability. This project could also go further in the direction of planner comparisons by testing these algorithms against a wider array of candidate planners. Finally, since the major advantage of using probabilistic planners is their ability to be easily modified for dynamical systems, I am interested in exploring the strengths and weaknesses of these planners for more complex systems.

References

[1] LaValle, Steven M. "Rapidly-exploring random trees: A new tool for path planning." (1998).

[2] Karaman, Sertac, and Emilio Frazzoli. "Sampling-Based Algorithms for Optimal Motion Planning." The International Journal of Robotics Research, vol. 30, no. 7, June 2011, pp. 846–89.

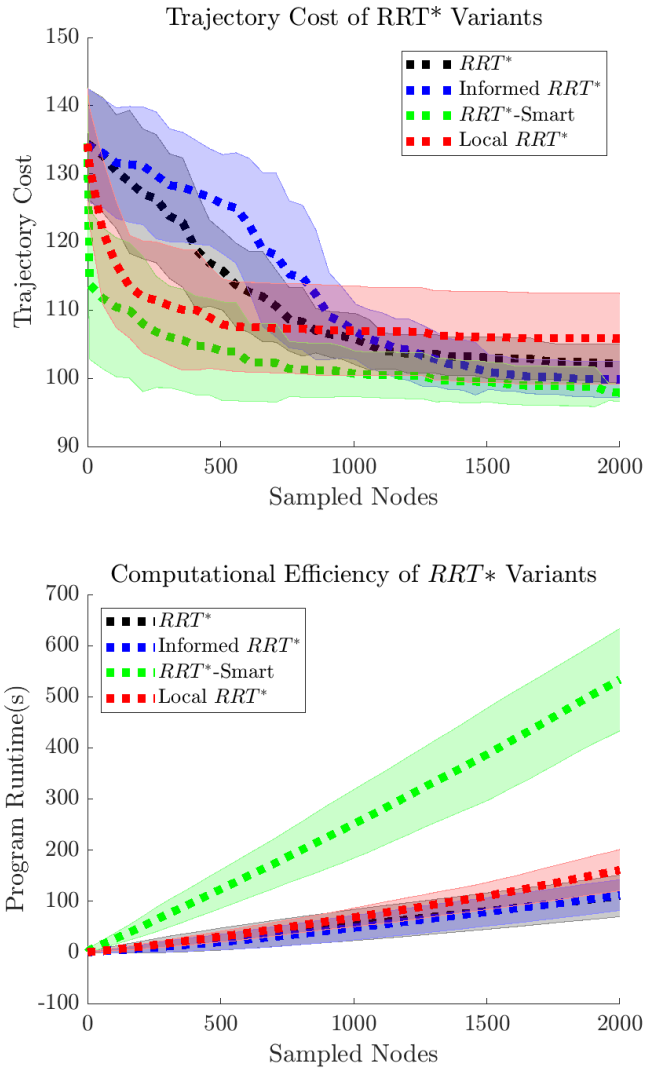


Fig. 12: Map 4 trajectory cost and computational efficiency for RRT* and its variants.

[3] Nasir, J., Islam, F., Malik, U., Ayaz, Y., Hasan, O., Khan, M., Muhammad, M. S. (2013). RRT*-SMART: A Rapid Convergence Implementation of RRT*. International Journal of Advanced Robotic Systems.

[4] Gammell, Jonathan D., Siddhartha S. Srinivasa, and Timothy D. Barfoot. "Informed RRT*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic." 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE, 2014.

[5] Glassman, Elena, and Russ Tedrake. "A Quadratic Regulator-based Heuristic for Rapidly Exploring State Space." IEEE International Conference on Robotics and Automation (ICRA), 2010

[6] Reist, P., Preiswerk, P., Tedrake, R. (2016). Feedback-motion-planning with simulation-based LQR-trees. The International Journal of Robotics Research, 35(11), 1393–1416.

[7] LaValle, Steven M., and James J. Kuffner Jr. "Randomized kinodynamic planning." The international journal of robotics research 20.5 (2001): 378-400.

[8] Schmitzberger, Erwin, et al. "Capture of homotopy classes with probabilistic road map." IEEE/RSJ International Conference on Intelligent Robots and Systems. Vol. 3. IEEE, 2002.